Preprint
UCRL-JC-138973

On the Design of a Parallel Object-Oriented Data Mining Toolkit

C. Kamath and E. Cantú-Paz

This article was submitted to
Workshop on Distributed and Parallel Knowledge Discovery
Boston, MA
August 20, 2000

May 17, 2000

U.S. Department of Energy
Lawrence
Livermore
National
Laboratory

On the Design of a Parallel Object-Oriented Data Mining Toolkit

Chandrika Kamath and Erick Cantú-Paz
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L-561
Livermore, CA 94551
kamath2,cantupaz1@llnl.gov

Abstract

As data mining techniques are applied to ever larger data sets, it is becoming clear that parallel processors will play an important role in reducing the turn around time for data analysis. In this paper, we describe the design of a parallel object-oriented toolkit for mining scientific data sets. After a brief discussion of our design goals, we describe our overall system design that uses data mining to find useful information in raw data in an iterative and interactive manner. Using decision trees as an example, we illustrate how the need to support flexibility and extensibility can make the parallel implementation of our algorithms very challenging. As this is work in progress, we also describe the solution approaches we are considering to address these challenges.

1 Introduction

Parallel data mining is the exploitation of fine grained parallelism in data mining, using tightly-coupled processors connected by a high-bandwidth interconnection network [6]. Implicit in this definition is the assumption that all the data used in mining is locally available, not globally distributed. This is often the case when commercial or scientific data is collected at one location, and often analyzed at the same location. If the size of the data is very large or a fast turnaround is required, it may be appropriate to mine the data using a parallel system. With 2-16 processor, Intel-based systems becoming inexpensive and common-place, the compute power necessary to implement this fine-grained parallelism is readily available.

Local data can be mined using either tightly- or loosely-coupled processors. In both cases, we need to focus on minimizing the communication costs across the processors. However, for loosely-coupled processors, this commmunication cost is typically much larger and may suggest the use of distributed data mining techniques, where the data is globally distributed, and communication done via the internet.

In this paper, we discuss the issues involved in designing and implementing an object oriented framework for mining data using tightly-coupled processors. Our focus is on distributed memory achitectures where each compute node has its own memory, and the nodes share only the interconnect. The architecture of such systems is scalable with increasing number of processors, making them well suited to mining massive data sets. We will also consider

the case where each node of a distributed memory system is a symmetric multi-processor (SMP), that is, the system is a cluster of SMPs.

The outline of this paper is as follows: In Section 2, we first outline our view of data mining in light of the challenges we face in mining scientific data. Next, we describe the system architecture we have designed for Sapphire, a large-scale data mining project at the Lawrence Livermore National Laboratory [13]. In Section 3, we use decision trees as an example to illustrate the problems we face as our need for flexibility meets the realities of parallel implementation. As this is work in progress, we discuss the solution approaches we are exploring to address these problems. Finally, in Section 4, we conclude with a summary.

2 The Data Mining Process

While there is broad agreement on what constitutes data mining, the tasks that are performed depend on the problem domain, the problem being solved, and the data. The Sapphire toolkit described in this paper is targeted to problems arising mainly from scientific applications, where the data is obtained from observations, experiments, or simulations. Scientific data analysis, while varied in scope, has several common challenges:

- **Feature extraction from low-level data:** Science data can be either image data from observations or experiments, or mesh data from computer simulations of complex phenomena, in two and three dimensions, involving several variables. This data is available in a raw form, with values at each pixel in an image, or each grid point in a mesh. As the patterns of interest are at a higher level, additional features must be extracted from the raw data prior to pattern recognition.

- **Noisy data:** Scientific data, especially data from observations and experiments, is noisy. This noise may vary within an image, from image to image, and from sensor to sensor. Removing the noise from data, without affecting the signal is a challenging problem in scientific data sets.

- **Size of the data:** Our data sets range from moderate to massive, with the smallest being measured in hundreds of Gigabytes and the largest a few Terabytes. As more complex simulations are performed, the data is expected to grow to the Petabyte range.

- **Need for data fusion:** Frequently, scientific data is collected from various sources, using different sensors. In order to use all available data to enhance the analysis, we need data fusion techniques. This is a non-trivial task if the data was collected at different resolutions, using different wavelengths, under different conditions.

- **Lack of labeled data:** Labeled examples in scientific data are usually generated manually. This tedious process is made more complicated as not all scientists may agree on a label for an object, or want the data mining algorithm to identify "interesting" objects, not just objects that are similar to the training set.

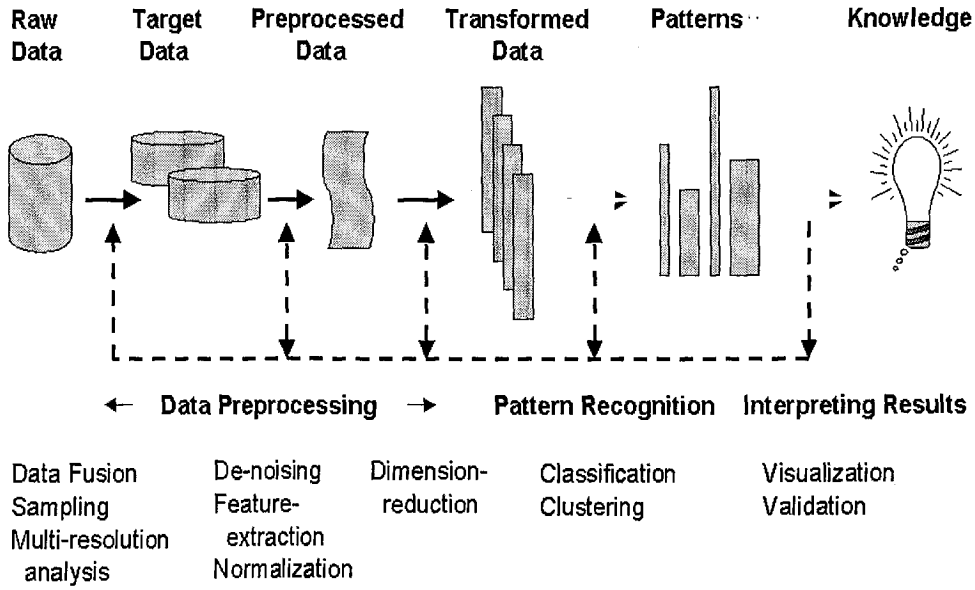

Figure 1: Data mining: an iterative and interactive process.

- **Data in flat files, not data bases:** Unlike commercial data, scientific data is rarely available in a cleaned state in data warehouses.
- **Mining data as it is being generated:** In the case of simulation data, scientists are interested in the behavior of the scientific phenomena as it changes with time. Sometimes, the time taken to output the result of the simulation at each time step may even exceed the simulation time itself. Since the simulations are run on large parallel computers, with hundreds to thousands of processors, it may be possible to perform some of the pre-processing while the data is being generated, resulting in a smaller output. While this idea seems simple, a practical implementation is non-trivial.

In light of these conditions, our definition of data mining starts with the raw data and includes extensive pre-processing (Figure 1). If the raw data is very large, we may use sampling and work with fewer instances, or use multiresolution techniques and work with data at a coarser resolution. This first step may also include data fusion, if required. Next, noise is removed and relevant features are extracted from the data. At the end of this step, we have a feature vector for each data instance. Depending on the problem and the data, we may need to reduce the number of features using dimension reduction techniques such as principal component analysis (PCA) or its non-linear versions. After this pre-processing, the data is ready for the detection of patterns. These patterns are then displayed to the user, who validates them appropriately.

The data mining process is iterative and interactive; any step may lead to a refinement of the previous steps. User feedback plays a critical role in the success of data mining in all stages, starting from the initial description of the data, the identification of potentially relevant features and the training set (where necessary), and the validation of the results.

2.1 The Sapphire System Design

In order to implement the data mining process in Figure 1 in a parallel setting, we need to put some thought into the design of the system. Our experience has shown that a good design should take into account the following:

- Not all problems require the entire data mining process, so each of the steps must be modular and capable of stand-alone operation.
- Not all algorithms are suitable for a problem, so the software should include several algorithms for each task, and allow easy plug and play of these algorithms.
- Each algorithm typically depends on several parameters, so the software should allow user friendly access to these parameters.
- Intermediate data must be stored appropriately to support refinement of the data mining process.
- The domain dependent and independent parts must be clearly identified to allow maximum re-use of software as we move from one application to another.

To accomodate these requirements, we put together the system architecture shown in Figure 2. The focus of Sapphire is on the compute-intensive tasks as these benefit the most from parallelism. Such tasks include decision trees, neural networks, image processing, and dimension reduction. Each class of algorithms is designed using object-oriented principles and implemented as a C++ class library. Parallelism is supported through the use of MPI and OpenMP for distributed and shared-memory parallel processing, respectively [7, 9]. We use domain-specific software for tasks such as reading, writing, and display of data. To support many different input data formats, such as FITS, View, and netCDF, we first convert each format into Sapphire's internal data format, prior to any processing. We are using RDB, a public-domain relational data base, as our permanent data store to store the intermediate data generated at each step. This has turned out to be invaluable as it has allowed us to experiment with different subsets of features and enabled us to easily support a growing data set. Our ultimate goal is that once we have each of the class libraries implemented, we will be able to provide a solution to a problem in a domain by simply linking the appropriate algorithms using a scripting language such as Python.

As we put together the system design for our object-oriented toolkit, we observed that two factors, unique to data mining, made it challenging to incorporate parallelism in the architecture:

- As data mining proceeds from feature extraction to the discovery of useful information, the data processed reduces in size. This reduction can be very drastic, e.g. from a Terabyte to a Megabyte. Furthur, some of the data pre-processing could occur on the parallel machine where the data is being generated, while the rest of the data analysis

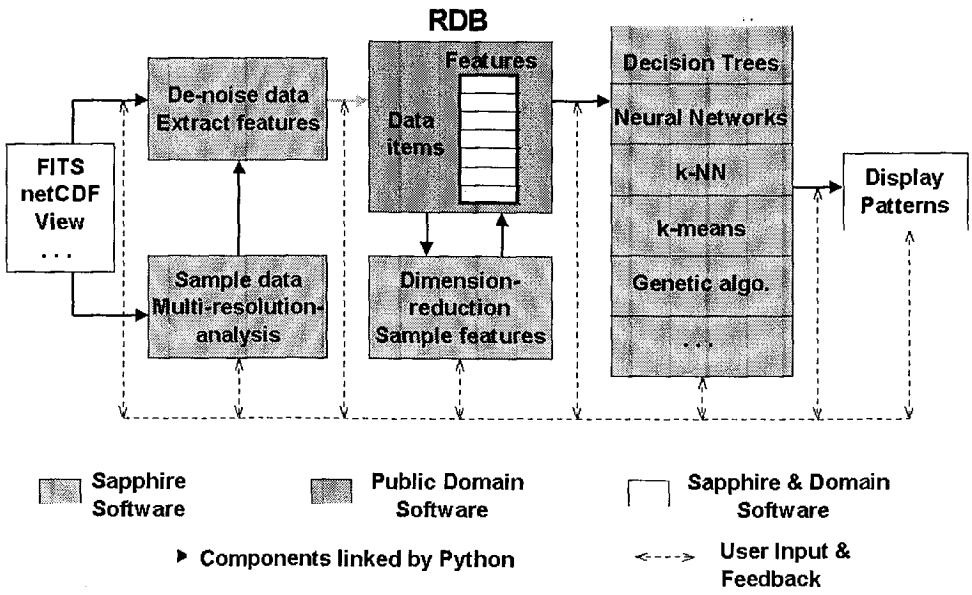

Figure 2: The Sapphire System Architecture: Flexible and Extensible

could take place on a different parallel machine with possibly fewer processors. Ensuring the end-to-end scalability of the data mining process under these circumstances could prove very challenging.

- The very nature of data mining requires close collaboration with the domain scientists at each step. Incorporating this iterative and interactive aspect into a parallel framework is a non-trivial task.

We next focus on one of the algorithms in data mining, namely, decision trees, describe our approach to the design and implementation of parallel software, and show how the need to support flexibility in a parallel implementation can give rise to conflicting requirements.

3 Parallel Decision Tree Software

Decision trees [2, 11, 10] belong to the category of classification algorithms wherein the algorithm learns a function that maps a data item into one of several pre-defined classes. Classification algorithms typically have two phases. In the training phase, the algorithm is "trained" by presenting it with a set of examples with known classification. In the test phase, the model created in the training phase is tested to determine how well it classifies known examples. If the results meet expected accuracy, the model can be put into operation to classify examples with unknown classification. This operation is embarrassingly parallel as several "copies" of the classifier can operate on different examples. It is important for the training phase of the classifier to be efficient as we need to find an optimum set of parameters which will enable accurate and efficient results during the operation of the classifier.

A decision tree is a structure that is either a leaf, indicating a class, or a decision node that specifies some test to be carried out on a feature (or a combination of features), with a branch and sub-tree for each possible outcome of the test. The decision at each node of the tree is made to reveal the structure in the data. Decision trees tend to be relatively simple to implement, yield results that can be interpreted, and have built-in dimension reduction. Parallel implementations of decision trees have been the subject of extensive research in the last few years [14, 16, 18, 15]. An approach used to construct a scalable decision tree was first described in the SPRINT algorithm [14]. Instead of sorting the features at each node of the tree as was done in earlier implementations, it uses a single sort on all the features at the beginning. The creation of the tree is thus split into two parts:

Initial Sorting

- First the training set is split into separate feature lists for each feature. Each list contains the identification (ID) number of the data instance, the feature value, and the class associated with the instance. This data is partitioned uniformly among the processors.
- Next, a parallel sort is performed on each feature list which results in each processor containing a static, contiguous, sorted portion of the feature. As a result of this sort, the data instances for one feature in one processor may be different from the data instances for another feature in the same processor. That is, all the features corresponding to one data instance may not belong to the same processor. This is the reason why the ID number of the data instance is included in the feature list.
- Next, we build count statistics for each of the features in each processor.

Creation of the decision tree

- Find the optimal split point:
  - Each processor evaluates each of the local feature lists to find the best local split (this is done in parallel by all processors).
  - It communicates the local best splits and count statistics to all processors.
  - Each processor determines the best global split (this is done in parallel by all processors).
- Split the data:
  - Each processor splits on the winning feature, and sends the ID numbers of its new left and right node data instances to all other processors.
  - Then, each processor builds a hash table containing all the ID numbers, and information on which instances belong to which decision tree node.
  - Next, each processor, for each feature, probes the hash table for each ID number to determine how to split that feature value.

- This process is carried out on the next unsolved decision tree node.

An improved version of the SPRINT algorithm that is scalable in both run-time and memory requirements is described in ScalParC [5]. This differs from SPRINT in two ways. First, a distributed hash table is used, instead of a single hash table which is replicated in each processor. This reduces memory requirements per processor, making the algorithm scalable with respect to memory. Second, as in SPRINT, the decision tree nodes are constructed breadth-first rather than depth-first and processor synchronization is held off until all work is done for that level of the tree. This not only limits the communication necessary for synchronization, but also results in better load balancing since processors that finish with one node of the tree can move directly on to the next node.

Our goal in the design and implementation of the Sapphire decision tree software is to take the ScalParC approach and extend it to include the following:

- **Support for several different splitting criteria:** The feature to test at each node of the tree, as well as the value against which to test it, can be determined using one of several measures. Depending on whether the measure evaluates the goodness or badness of a split, it can be either maximized or minimized. Let T be the set of n examples at a node that belong to one of k classes, and T_L and T_R be the two non-overlapping subsets that result from the split (that is, the left and right subsets). Let L_j and R_j be the number of instances of class j on the left and the right, respectively. Then, the split criteria we want to support include [8]:
  - Gini: This criterion is based on finding the split that most reduces the node impurity, where the impurity is defined as follows:

$$L_{Gini} = 1.0 - \sum_{i=1}^{k} (L_i/|T_L|)^2 \quad , \quad R_{Gini} = 1.0 - \sum_{i=1}^{k} (R_i/|T_R|)^2$$

$$\text{Impurity} = (|T_L| * L_{Gini} + |T_R| * R_{Gini})/n$$

    where $|T_L|$ and $|T_R|$ are the number of examples, and L_{Gini} and R_{Gini} are the Gini indices on the left and right side of the split, respectively. This criterion can have problems when there are a large number of classes.
  - Twoing rule: In this case, a "goodness" measure is evaluated as follows:

$$\text{Twoing value} = (|T_L|/n) * (|T_R|/n) * \left(\sum_{i=1}^{k} |L_i/|T_L| - R_i/|T_R|| \right)^2$$

  - Information gain: The information gain associated with a feature is the expected reduction in entropy caused by partitioning the examples according to the feature. Here the entropy characterizes the (im)purity of an arbitrary collection of

examples. For example, the entropy prior to the split in our example would be:

$$\text{Entropy}(T) = \sum_{i=1}^{k} -p_i \log_2 p_i \quad , \quad p_i = (L_i + R_i)/n$$

where p_i is the proportion of T belonging to class i and $(L_i + R_i)$ is the number of examples in class i in T. The information gain of a feature F relative to T is then given by

$$\text{Gain}(T, F) = \text{Entropy}(T) - \sum_{v \in \text{values}(F)} |T_v| * \text{Entropy}(T_v)/|T| \qquad (1)$$

where T_v is the subset of T for which the feature F has value v. Note that the second term above is the expected value of the entropy after T is partitioned using feature F. This is just the sum of the entropies of each subset T_v, weighted by the fraction of examples that belong to T_v. This criterion tends to favor features with many values over those with few values.

- Information gain ratio: To overcome the bias in the information gain measure, Quinlan [11] suggested the use of information gain ratio which penalizes features by incorporating a term, called the split information, that is sensitive to how broadly and uniformly the feature splits the data.

$$\text{Split Information}(T, F) = -\sum_{i=1}^{c} (|T_i|/n) \log_2 (|T_i|/n) \qquad (2)$$

where T_i are the subsets resulting from partitioning T on the c-valued feature F. Note that the split information is the entropy of T with respect to the values of the feature F. The Gain ratio is then defined as

$$\text{Gain Ratio}(T, F) = \text{Gain}(T, F)/\text{Split Information}(T, F) \qquad (3)$$

- Max Minority: This criterion is defined as

$$L_{minority} = \sum_{i=1, i \neq \max L_i}^{k} L_i \quad , \quad R_{minority} = \sum_{i=1, i \neq \max R_i}^{k} R_i$$

$$\text{Max minority} = \max(L_{minority}, R_{minority})$$

This has the theoretical advantage that a tree built by minimizing this measure will have depth at most $\log n$. This is not a significant advantage in practice and trees created by other measures are seldom deeper than the ones produced by Max Minority.

- Sum Minority: This criterion minimizes the sum of $L_{minority}$ and $R_{minority}$, which is just the number of misclassified instances.

- **Support for non-axis-parallel decision trees:** Traditional decision trees consider a single feature at each node, resulting in hyperplanes that are parallel to one of the axes. While such trees are easy to interpret, they may be complicated and inaccurate in the case where the data is best partitioned by an oblique hyperplane. In such instances, it may be appropriate to make a decision based on a linear combination of features, instead of a single feature. However, these oblique trees can be harder to interpret. They can also be more compute intensive as the problem of finding an oblique hyperplane is much harder than the problem of finding an axis-parallel one. None-the-less, our early research has shown that when used in conjunction with genetic algorithms, these oblique classifiers could prove competitive in some cases [4]. To further explore these ideas, we want to design our software such that, in addition to axis parallel trees, it can support the following types of splits at each node:

  - CART-LC: Brieman et. al, in [2], suggested the use of linear combinations of features to split the data at a node. If the features for a data instance are given as $(x_1, x_2, \ldots, x_n, c)$, where c is the class label associated with the instance, then, we search for a best split of the form

    $$\sum_{i=1}^{n} a_i x_i \leqslant d \quad \text{where} \quad \sum_{i=1}^{n} a_i^2 = 1 \tag{4}$$

    and d ranges over all possible values. The solution approach cycles through the variables $x_1, \ldots, x_n$, trying to find the best split on each variable, while keeping the others constant. A backward deletion process is then used to remove variables that contribute little to the effectiveness of the split. This approach is fully deterministic and can get trapped in a local minima.
  - OC1: The oblique classifier OC1 described in [8] attempts to address some of the limitations of the CART-LC approach by including randomization in the algorithm that finds the best hyperplane. Further, multiple random re-starts are used to escape local minima. In order to be at least as powerful as the axis-parallel decision trees, OC1 first finds the best axis-parallel split at a node before looking for an oblique split. The axis-parallel split is used if it is better than the best oblique split determined by the algorithm for that node. Note that it does not make much sense to use an oblique split when the number of examples at a node is approximately the same as the number of features as the data then underfits the concept to be learned. In such cases, OC1 shifts to an axis-parallel split when the number of examples falls below a user-specified threshold.
  - OC1-GA: It is possible to use evolutionary algorithms to solve the optimization problem in the creation of oblique classifiers. In OC1-GA, we find the best hyperplane represented by the coefficients $(a_1, \ldots, a_n, d)$ using genetic algorithms. The concatenated version of these coefficients forms a member of the population. The fitness of each member is determined by evaluating how well it splits the examples at a node for a given split criterion. Genetic algorithms thus allow us to change all the coefficients at a time instead of a series of univariate splits considered in OC1 and CART-LC.

We have explored two options for evolutionary algorithms. In one case we use a (1+1) evolutionary strategy with adaptive mutations. The initial hyperplane is the best axis-parallel split for the node. For each hyperplane coefficient, we have a mutation coefficient, which is updated at each iteration and used to determine the new hyperplane coefficient. We then select the best between the parent and child hyperplanes. In the second approach, we use a simple generational GA with real valued genes. The initial population consists of 10% copies of the axis-parallel hyperplane, and the rest are generated randomly. Our initial experiments have shown that in some cases, the OC1-GA approaches are faster and more accurate than OC1 [4].

- **Support for both numeric and nominal features.**
- **Support for different pruning options and stopping criteria:** We are interested in exploring different ways to avoid over-fitting through pruning and rules that decide when to stop splitting, such as the cost complexity pruning technique of Breiman [2] or the minimum description length approach suggested by Quinlan and Rivest [12].

Our main challenge is to support these options and include the flexibility to add new options without re-writing the code that supports the parallel implementation of the decision tree.

3.1 The Sapphire Decision Tree Design

As explained in the previous section, we are interested in a decision tree design that gives us enough flexibility to experiment with different options within a parallel implementation. It is relatively easy to support some of these options within the context of an object-oriented design. For example, different splitting criteria can be easily supported by having an abstract base class from which concrete classes for the split criterion are inherited. These concrete classes implement the function used to determine the quality of a split. The user can then instantiate an object in one of these classes to indicate the split criterion used at all nodes of the tree. This choice would be communicated to the decision tree object by passing a pointer to the base split criteria class as an argument. A similar situation holds in the case of pruning options which are executed after the tree is built. In both cases, the main operation performed by the class is at a low-enough level that no parallelism is required in the implementation of the operation.

The UML class diagram [17] for our decision tree design is given in Figure 3. The prefix di_ is used to indicate classes that contain domain information, tbox_ to indicate toolbox classes for general use, and dt_ to indicate classes used in the decision tree. Note that the di_ classes can be used in other classification and clustering algorithms, not just decision trees. A brief description of the classes is as follows:

- di_FeatureValue: This contains either a nominal (discrete) feature or a numeric (continuous) feature, but never both at the same time.

- di_InstanceInfo: This contains the number of features, the name of the features and their type for a data instance.

- di_Instance: This contains the features for a data instance. It is typically used in conjunction with di_InstanceInfo.

- di_InstanceArray: This can be used for the training set, where each data instance has several features or even for the feature lists that contain only a single feature and are created in the first part of the parallel decision tree.

- tbox_NominalHistogram: This creates a histogram for nominal data.

- dt_SplitCriterion: This abstract base class represents the criterion to be used to evaluate the split at each node. The derived classes denote the value that is returned after an evaluation of a split. As we find new ways of judging a split, a new class can be derived from the base class to implement that split criterion. The same split criterion is used in the entire decision tree.

- dt_SplitFinder: This base class represents the approach used to find the split - whether axis-parallel, oblique, CART-LC etc. Derived classes implement the actual determination of the split. The SplitFinder used at any node of the tree may vary depending on several factors. For example, if the instances at a node are few, an axis parallel approach may be chosen instead of an oblique one. Or, evaluation of an oblique split may indicate that an axis-parallel split is a better choice for the data at a node. Regardless of the choice of SplitFinder, the user can independently select the split criterion used to evaluate the split. It is possible to exploit parallelism within the SplitFinder class.

- dt_TreeNode: This class contains the information on a node of the tree. It includes pointers to the InstanceArrays stored using a single feature at a time, the left- and right-hand sides of the split made at the node, the type of SplitFinder, the count statistics for each feature, and pointers to the children nodes created by the split. Once the split is determined using the SplitFinder, the TreeNode object is responsible for actually splitting the instances among the children node. Parallelism can be exploited within this class.

- dt_DecisionTree: This is the main class that creates, tests, and applies the tree. It can also print out the tree, save it to a file, and read it back from a file. Starting with a root TreeNode that contains the entire training set, it creates the child nodes by choosing the appropriate SplitFinder, using the SplitCriterion set by the user. The single sort that is required by the parallel implementation is done at the beginning of the training of the decision tree. Parallelism is exploited within this class.

One of the challenges we face in supporting several different options in parallel decision tree software is that the approach taken for efficient implementation of one option could directly conflict with the efficient implementation of another option. An interesting case of this arises in the SplitFinder class. The ScalParC approach, which generates axis-parallel trees,

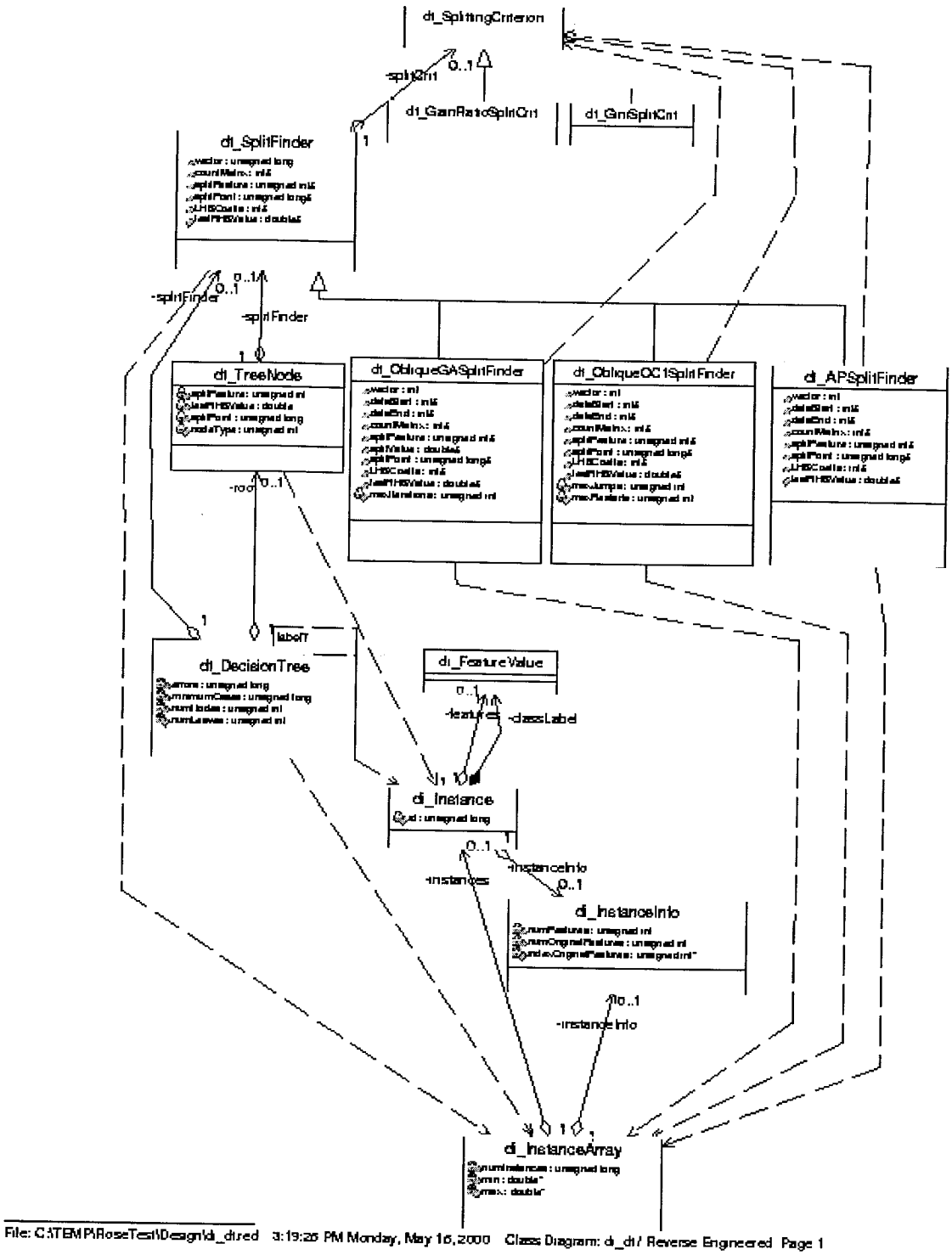

Figure 3: The UML Class Diagram for Decision Trees

sorts each feature at the beginning of the creation of the tree. As mentioned earlier, this results in the features that comprise a single data instance to be spread across more than one processor. However, for oblique classifiers, in order to evaluate a split, all features in a data instance are needed. If these features are spread across processors, communication would be required. This communication could very likely have an irregular pattern and, depending on how the features corresponding to a data instance are spread our among the processors, could be extensive. This would suggest that to support oblique splits, we should not sort each of the features prior to the creation of the decision tree. However, regardless of the technique used to calculate an oblique split, we still need to evaluate axis-parallel splits. For example, an oblique split starts with an axis parallel split, is compared with an axis parallel split in order to select the better of the two, and determines an axis-parallel split for each coefficient at a time, keeping the others constant.

This gives rise to an interesting dilemma - should we sort each feature at the beginning or not? It is always possible to have two sets of features, one sorted and the other unsorted, even though it would almost double the memory requirements. The other option is to work with only one set of features, but should we picked the sorted or the un-sorted one? Since sorting would result in extensive communication in the case of oblique splits, a possible solution approach would be to see if we could somehow mimic the axis-parallel split efficiently on un-sorted data.

To determine the best axis parallel split, we first sort the values for a feature, and then determine the value of a split if the split point was taken mid-way between two consecutive feature values. The best split across all features is chosen as the best split at a node. Instead of this approach, suppose we generate a histogram for each of the features, we can select as a split value the boundary value of each bin in the histogram. If the histogram kept track of the count statistics for each class in a bin, we could use this information to select the best split based on any splitting criterion. If the bin widths are chosen appropriately, this could give a good approximation to the axis-parallel split. Related work is described in [1].

A different issue we need to address in the parallelization of decision trees is the implementation on clusters of SMPs, where we may need to use both distributed and shared memory programming. This could be most beneficial in the case where we use genetic algorithms to search for the best oblique hyperplane, as genetic algorithms tend to be expensive to implement. This would give rise to some interesting solution approaches. Suppose the data instances with unsorted features are distributed uniformly across the nodes of a parallel system. Then the SMP processors within each node could work on finding the best oblique hyperplane for its set of data instances, while occasionally exchanging members with other nodes in order to find a hyperplane that best splits the entire set of data instances [3].

We are in the midst of our experimentation with the solution approaches described above. We will report progress on our work during the KDD workshop on Distributed and Parallel Knowledge Discovery.

4 Summary

In this paper, we have discussed our design goals for a parallel object-oriented software toolkit for mining scientific data sets. We have described how our design can meet the diverse needs of our applications. Focusing on a specific example, namely decision trees, we presented some of the challenges we face as we implement several different variants of decision trees within a parallel framework. We have also briefly described our approach to addressing these challenges.

5 Acknowledgements

We acknowledge the contributions of the rest of the Sapphire project team: Chuck Baldwin, Imola Fodor, and Nu Ai Tang.

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

[1] ALSABTI, K., RANKA, S., AND SINGH, V. CLOUDS: A decision tree classifier for large datasets. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining* (1998).

[2] BREIMAN, L., FRIEDMAN, J., OLSHEN, R. A., AND STONE, C. *Classification and Regression Trees.* CRC Press, Boca Raton, Florida, 1984.

[3] CANTÚ-PAZ, E. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis 10*, 2 (1998), 141–171.

[4] CANTÚ-PAZ, E., AND KAMATH, C. Using evolutionary algorithms to induce oblique decision trees. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), July 2000* (2000).

[5] JOSHI, M. V., KARYPIS, G., AND KUMAR, V. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proceedings of the 12th International Parallel Processing Symposium* (1998).

[6] KAMATH, C., AND R., M. Scalable data mining through fine-grained parallelism: the present and the future. In *Advances in Distributed and Parallel Knowledge Discovery*, H. Kargupta and P. Chan, Eds. AAAI/MIT Press, 2000.

[7] MPI Forum. *http://www.mpi-forum.org.*

[8] MURTHY, K. V. S. *On Growing Better Decision Trees from Data.* PhD thesis, Johns Hopkins University, 1997.

[9] OpenMP application program interface. *http://www.openmp.org.*

[10] QUINLAN, J. *C4.5: Programs for Machine Learning.* Morgan Kaufman, San Mateo, California, 1993.

[11] QUINLAN, J. R. Induction of decision trees. *Machine Learning 1* (1986), 81–106.

[12] QUINLAN, J. R., AND RIVEST, R. Inferring decision trees using the minimum description length principle. *Information and Computation 80*, 3 (1989), 227–248.

[13] Sapphire: Large-Scale Data Mining and Pattern Recognition. *http://www.llnl.gov/casc/sapphire.*

[14] SHAFER, J., AGRAWAL, R., AND MEHTA, M. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the 22nd Conference on Very Large Data Bases* (1996).

[15] SREENIVAS, M., K., A., AND RANKA, S. Parallel out-of-core decision tree classifiers. In *Advances in Distributed and Parallel Knowledge Discovery*, H. Kargupta and P. Chan, Eds. AAAI/MIT Press, 2000.

[16] SRIVASTAVA, A., HAN, E. H., KUMAR, V., AND SINGH, V. Parallel formulations of decision-tree classification algorithms. *To appear in Data Mining and Knowledge Discovery; An International Journal* (1999).

[17] Object Management Group: The Unified Modeling Language. *http://www.omg.org.*

[18] ZAKI, M. J., HO, C. T., AND AGRAWAL, R. Parallel Classification on SMP Systems. In *Proceedings of the 1st Workshop on High Performance Data Mining* (1998).